Final Report

Anomaly Detection At Multiple Scales (ADAMS)

Sponsored by the

Defense Advanced Research Projects Agency (DOD)[1]

Issued by

U.S. Army Aviation and Missile Command Under

Contract No. W31PQ-11-C-0229

November 9, 2011

Name of Contractor: Allure Security Technology Inc.
Principal Investigator: Hugh Thompson
Business Address: 5 Penn Plaza, 23^{rd} Floor, New York, NY 10001-1810
Phone: 201-444-6944

Effective Date of Contract: April 14, 2011
Short Title of Work: ADAMS Behavioral Sensors

Contract Expiration Date: December 31, 2013
Reporting Period: April 14, 2011 – October 13, 2011

CAGE Code: 5KE34
Issue by DODACC: W31P4Q
Admin by DODAAC: S3309A
Acceptor DODACC: W90BWX
Service Approver: W90BWX

DARPA Program Manager: Rand Waltzman, Rand.Waltzman@darpa.mil

[1]The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either express or implied, of the Defense Advanced Research Project Agency or the U.S. Government.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
10-11-2011	Final Project Report	14-04-2011 - 13-10-2011

4. TITLE AND SUBTITLE
ADAMS Behavioral Sensors

5a. CONTRACT NUMBER
W31PQ-11-C-0229

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
ARPA Order BK21-00

6. AUTHOR(S)
Hugh Thompson
Salvatore J. Stolfo
Angelos D. Keromytis
Shlomo Hershkop

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Allure Security Technology Inc
5 Penn Plaza
23rd Floor
New York, NY 10001-1810

8. PERFORMING ORGANIZATION REPORT NUMBER
CLIN 0001AA

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency (DOD)
IIO

10. SPONSOR/MONITOR'S ACRONYM(S)
DARPA IIO

11. SPONSORING/MONITORING AGENCY REPORT NUMBER

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The recent disclosure of sensitive and classified government documents through WikiLeaks demonstrates a new systemic threat, exfiltration and broad global broadcast of government confidential data and information. Allure Security Technology Inc., a Columbia University spinout company, is developing techniques and mechanisms to identify likely malicious insiders by leveraging automatically generated misinformation and system and network monitoring technologies such as Data Leak Prevention (DLP). We are developing a baseline system that will demonstrate the feasibility of identifying specific types of insiders by developing a prototype for automatically generating and distributing believable misinformation based upon operator-defined templates, and then tracking access and attempted misuse of it. We call this "disinformation technology", FOG computing.

15. SUBJECT TERMS
Anomaly Detection at Scale, Behavioral Sensors, FOG Computing.

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Hugh Thompson
					19b. TELEPONE NUMBER *(Include area code)* 201-444-6944

Contents

1 Introduction

This is the design documents for the Allure Defender system. This document is a high level design and API of the components that make up the Allure Defender system. We outline all the high-level pieces and then the individual components, their behaviors, expected input/outputs, and relationships. We will discuss specific implementation and design choices and languages and libraries that will be used. In addition we will cover specific user cases and illistrate some running examples. Last we refer to a running system which impliments many of the components we cover in the document.

The goal of the document is for a designer to create a working system and or verify a working system conforms to the specifications outlined in the document.

1.1 Pointers to Phase I Task Deliverables

We point to the various sections of the document that describe the work we have done as part of our Phase I effort.

- Task 1: Conduct research on characteristics of Document-based Behavioral Sensors (DBS): Sections 2 and 3.
- Task 2: Document generation engine design: Sections 2 and ?? to 14 (inclusive).
- Task 3: Per-document type feasibility analysis: Sections 3, 11 and 12.
- Task 4: Source code plug-in design: Section 6.
- Task 5: Beacon analysis: Sections 3 and 11 and .
- Task 6: Beacon generator and sensor design: Sections 11 and 12.
- Task 7: Search behavior modeling for host-based sensor design: Section 5.
- Task 8: Host-based sensor design: Section 4.
- Final Deliverable: this whole document.

2 System Architecture and Design

The purpose of Allure Defender is to create behavioral sensors that can be used to detect the presence (and possibly other attributes) of a malicious, stealthy (and possibly "insider") adversary. The behavioral sensors used in the first version of Allure Defender will consist of enticing documents, combined with a number of detection techniques. Thus, at a very high level, the system is divided into two components: *document generation* and *misbehavior detection.*

The **document generation** component will create documents (henceforth referred to as *Decoy Documents*, or *DD* for short) in various formats (e.g., Word, Excel, PDF, Powerpoint, email messages, Instant Messaging logs, ...) that contain one of several features :

- a *"mark"* allowing Allure Defender to determine whether a file is a DD, and possibly allow legitimate users to avoid accessing/triggering the DD;
- one or more *"beacons"*, which will cause the application processing the DD to emit some sort of discernible signal;
- Enticing Information (henceforth referred to as *EI*) which, if acted upon by the adversary, will allow detection. Such information includes URLs (for various protocols), account information (e.g., username/password), and others that may be developed in the future; and
- Enticing Content (henceforth referred to as *EC*) that will attract the adversary to the DD (e.g., if they are using a search function) without raising suspicion, will support the presence of the EI in the document, and will allow the DDs to "fit in" with the rest of the environment on which they have been deployed.

DDs may be deployed on servers, databases, user desktops and laptop, mobile devices, honeypots, or other locations. It is desirable that all of these seeding techniques be supported.

The EC may be generated based on templates, synthesized from private sources (e.g., by mining existing documents at the directory/account/system/server to be seeded), synthesized from public sources (e.g., documents acquired through search engines) based on high-level templates, or synthesized from

Figure 1: High Level Architecture

public sources using information mined from existing documents at the directory/account/system/server to be seeded. Any combination of these techniques may be used to generate DDs, and a specific DD may be the result of several such techniques being used simultaneously.

The *misbehavior detection component* consists of a variety of subcomponents, some of which are specific to the beacon techniques used:

- honeypot servers, pointed to by URLs and similar information;
- intrusion detection systems combined with legitimate servers/services, when the latter can be used for detection purposes without compromising primary functionality (e.g., invalid username/password login attempts, specific directories in a filesystem or web server hierarchy, DNS server queries, and so on);
- Data Leakage Prevention (DLP) subsystems, which may operate at various points in the system, e.g., network, filesystem, memory, and others. The DLP may be a priori aware of the identity and location of the DDs, or it may be able to identify them on the fly via the "mark".

The design of the architecture (see Figure 1) attempts to cleanly divide the functionality of the different subsystems into self-managing components allowing maximum flexibility of the system to adopt to changes while allowing all the components to seamlessly work together. The design reflects the

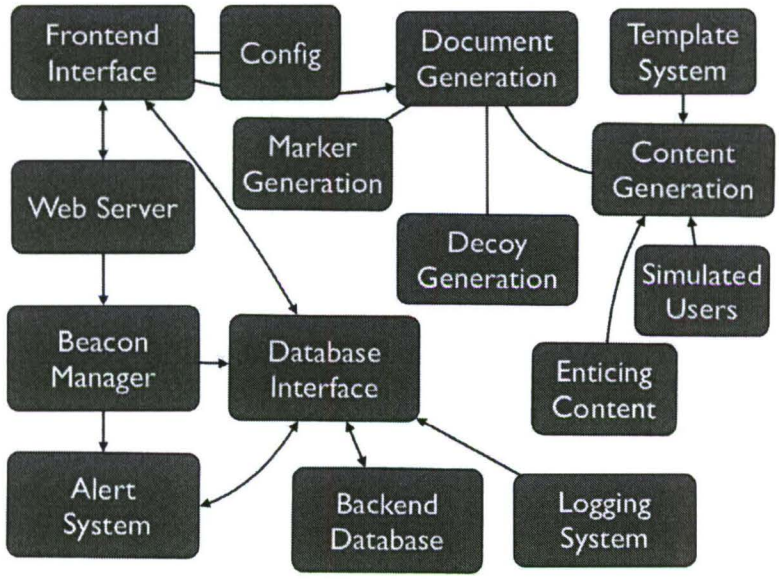


Figure 2: Component Architecture

facts that (*a*) documents may be requested via different interfaces (e.g., web-server front-end, client-side logic interacting over the network, client-side application with generation library, and possibly others); (*b*) the documents may contain a combination of enticing information, marks and beacons, based on the desired configuration, (*c*) the corresponding detection capabilities can vary (and should be extensible so that we can add further capabilities as future research directs), and (*d*) the documents, and specifically the enticing content, may be generated through a variety of means. Examples of the latter include:

- Template-based documents, with "fill-in the blanks" generation based on random (but realistic) information.
- As above (template-based generation), but the inserted information may be retrieved from the organization's own information (e.g., employee directory), from the user machines (e.g., based on information gleaned from email records or from other — possibly similar — files on the machines where the documents will be deployed on), from social network public data, etc.

- As above (template-based generation with personalized inserted information), where the templates are directly derived from suitable existing documents in the organization or user machine where the DDs will be deployed (e.g., files which already contain username/password combinations, which can be modified).

- As above, where the templates are abstracted from suitable existing documents, e.g., by copying the structure of existing documents and inserting text (either automatically generated or retrieved from public or private/organizational sources) that shares keywords with the corresponding text in the original document.

- As above, where the structure is a combination of the structure of similar-but-not-identical documents that already exist in the target environment.

- As above, where the templates are abstracted from suitable existing documents by copying only their structure, but the text can be based on independently produced (e.g., configuration-derived) keywords.

We envision a simple system at first (template-based), with more powerful capabilities integrated over time. Figure 2 shows our component architecture at a greater level of detail. The various elements shown there include:

- Webserver: software platform running the system;

- Frontend Interface: forward-facing GUI for accounts, documents, and maintainance/administration;

- Backend Database: database system which stores/retrieves user, decoy activities;

- Document Generation: system for creating and managing the decoy documents;

- Decoy generation: technology for creating decoys which can be tracked;

- Content System: set of technologies for managing and manipulating believable content to make the documents enticing;

- Simulated Users: the component handling a collection of decoy users for use in document system (to create a timeline of documents/interactions);

- Marker Generation: the creation of markers, inserted into decoy documents, that can be tracked by DLP or other host-based sensors;
- Template System: the store of decoy templates, from which decoy documents are generated;
- Beacon Manager: the system component that detects events triggered by document beacons;
- Logging System: the component the detects the attempted use of enticing information, and which may be subdivided into multiple subcomponents and distributed across the organization and outside resources;
- Alert System: the component responsible for notifying users and/or administrators about beacon, marker and EI events.

We begin by discussing the high-level properties of decoy documents and beacons. We then discuss our system design in more detail. But first, we will give a brief overview of related work in this space and our threat model.

2.1 Related Work

The use of deception, or decoys, plays a valuable role in the protection of systems, networks, and information. The first use of decoys (*i.e.,* in the cyber domain) has been credited to Cliff Stoll [31, 24] and detailed in his novel "The Cuckoos Egg" [25], where he provides a thorough account of his crusade to catch German hackers breaking into Lawrence Berkeley Laboratory computer systems. Stoll's methods included the use of bogus networks, systems, and documents to gather intelligence on the German attackers who were apparently seeking state secrets. Among the many techniques waged, he crafted "bait" files, or in his case, bogus classified documents that really contained non-sensitive government information and attached "alarms" to them so that he would know if anyone accessed at them. To Stoll's credit, a German hacker was eventually caught and it was found that he had been selling secrets to the KGB.

Deception-based information resources that have no production value other than to attract and detect adversaries (like those used by Stoll) are commonly known as Honeypots [11]. Honeypots serve as effective tools for

profiling attacker behavior and to gather intelligence to understand how attackers operate. Honeypots are considered to have low false positive rates since they are designed to capture only malicious attackers, except for perhaps an occasional mistake by innocent users. Spitzner described how honeypots can be useful for detecting insider attack [23], in addition to the common external threats for which they are traditionally known. He discusses the use of honeytokens, which he defines as "a honeypot that is not a computer" [24], citing examples that include bogus medical records, credit card numbers, and credentials, with descriptions of how they can be used to detect malicious insiders [23, 24]. In a similar spirit, Webb *et al.* [28] showed how honeypots can be useful for detecting spammers. Although spam is not the focus of this work, their deceptive approach to detecting it may be applicable. In current systems, the decoy/honeytoken creation is a laborious and manual process requiring large amounts of administrator intervention. In contrast, we propose the seeding of decoy information (of various different types) throughout an operational system. Our work extends these basic ideas to an automated system of managing the creation and deployment of these honeytokens.

Yuill *et al.* [31] extend the notion of honeytokens with a "honeyfile system" to support the creation of bait files, or as they define them, "honeyfiles." The honeyfile system is implemented as an enhancement to the Network File Server. The system allows for any file within user file space to become a honeyfile through the creation of a record associating a filename to userid. The honeyfile system monitors all file access on the server and alerts users when honeyfiles have been accessed. Their work does not focus on the content or automatic creation of files, but they do elicit some of the challenges of creating deceptive files (with respect to names) that we address.

We introduce a set of properties of decoys to guide their design and maximize the deception they induce for different classes of insiders who vary by their level of knowledge and sophistication. To the best of our knowledge, the synthesis of these properties is indeed novel a contribution. Bell and Whaley [2] have described the structure of deception as a process of hiding the real and showing showing the false. They introduce several methods of hiding that include masking, repackaging, and dazzling, along with three methods of showing that include mimicking, inventing, and decoying. Yuill *et al.* [30] expand upon this work and characterize deceptive hiding in terms of how it defeats an adversary's discovery process. They describe an adversary's discovery process as taking three forms: direct observation, investigation based on evidence, and learning from other people or agents. Their work offers a

process model for creating deceptive hiding techniques based on how they defeat an adversary's discovery process.

The decoy documents we introduce utilize similar deception mechanisms as well as beacons to signal a remote detect and alert in real-time time when a decoy has been opened. Web bugs are a class of silent embedded tokens which have been used to track usage habits of web or email users [18]. Unfortunately, they have been most closely associated with unscrupulous operators, such as spammers, virus writers, and spyware authors who have used them to violate users privacy. Typically they will be embedded in the HTML portion of an email message as a non-visible white on white image, but they have also been demonstrated in other forms such as Microsoft Word, Excel, and PowerPoint documents [21]. When rendered as HTML, a web bug triggers a server update which allows the sender to note when and where the web bug was viewed. Animated images allow the senders to monitor how long the message was displayed. The web bugs operate without alerting the user of the tracking mechanisms. The advantage for legitimate advertisers is that this allows them to monitor advertisement effectiveness, while privacy advocates worry that this technology can be misused to spy on users' habits. Our work leverages the same ideas, but extends them to other document classes and is more sophisticated in the methods used to draw attention. In addition, our targets are insiders who should have no expectation of privacy on a system they violate.

2.2 Threat Model

The insider seeks to identify and avoid the decoys and abscond with "real" information. We broadly define four monotonically increasing levels of insider sophistication and capability. Some will have tools available to assist in deciding what is a decoy and what is real. Others will only have their own observations and thoughts.

- **Low:** Direct observation is the only tool available. The adversary largely depends on what can be gleaned from a first glance. We strive to defeat this level of adversary with our beacon documents, even though decoys with embedded beacons may be distinguished with more advanced tools.

- **Medium:** A more thorough investigation can be per- formed by the insider; decisions based on other, possibly outside evidence, can be

made. For example, if a decoy document contains a decoy account credential for a particular identity, an adversary may verify that the particular identity is real or not by querying an external system (such as www.whitepages.com). Such adversaries will require stronger decoy information possibly corroborated by other sources of evidence.

- **High:** Access to the most sophisticated tools are available to the attacker (e.g., super computers, other informed people who have organizational information). The notion of the "Perfect Decoy" described in the next section may be the only indiscernible decoy by an adversary of such caliber.

- **Highly Privileged:** Probably the most dangerous of all is the privileged and highly sophisticated user. Such attackers might even be aware that the system is baited and will employ sophisticated tools to try to analyze, disable, and avoid decoys entirely. As an example of how defeating this level of threat might be possible, consider the analogy with someone who knows encryption is used (and which encryption algorithm is used), but still cannot break the system because they do not have knowledge of an easy-to-change operational parameter (the key). Likewise, just because someone knows that decoys are used in the system does not mean they should be able to identify them. This is the principal- coming up with a scheme to satisfy it remains an open problem.

3 Decoy Document Properties

We enumerate various properties and means of measuring these properties that are associated with decoy documents to ensure their use will be likely to snare an inside attacker. We introduce the following notation for these definitions.

Believable[1]: Capable of eliciting belief or trust; capable of being believed; appearing true; seeming to be true or authentic.

A good decoy should make it difficult for an adversary to discern whether they are looking at an authentic document from a legitimate source or if they are indeed looking at a decoy. We conjecture that believability of any particular decoy can be measured by adversary's failure to discern one from the other. We formalize this by defining a decoy believability experiment. The experiment is defined for the document space M with the set of decoys D such that $D \subseteq M$ and $M - D$ is the set of authentic documents.

The Decoy Believability Experiment: $\mathbf{Exp}^{believe}_{A,D,M}$

- For any $d \in D$, choose two documents $m_0, m_1 \in M$ such that $m_0 = d$ or $m_1 = d$, and $m_0 \neq m_1$; that is, one is a decoy we wish to measure the believability of and the second is chosen at random from the set of authentic documents.
- Adversary A obtains m_0, m_1 and attempts to choose $\hat{m} \in \{m_0, m_1\}$ such that $\hat{m} \neq d$, using only information intrinsic to m_0, m_1.
- The output of the experiment is 1 if $\hat{m} \neq d$ and 0 otherwise.

For concreteness, we build upon the definition of "Perfect Secrecy" proposed in the cryptography community [13] and define a "perfect decoy" when:

$$\Pr[\mathrm{Exp}^{believe}_{A,D,M} = 1] = 1/2$$

[1]For clarity, each property is provided with its definition gleaned from online dictionary sources.

The decoy is chosen in a believability experiment with a probability of 1/2 (the outcome that would be achieved if the volunteer decided completely at random). That is, a perfect decoy is one that is completely indistinguishable from one that is not. A benefit of this definition is that the challenge of showing a decoy to be believable, or not, reduces to the problem of creating a "distinguisher" that can decide with probability better than 1/2.

In practice, the construction of a "perfect decoy" might be unachievable, especially through automatic means, but the notion remains important as it provides a goal to strive for in our design and implementation of systems. For many threat models, it might suffice to have less than perfect believable decoys. For our proof-of-concept system described below, we generate receipts and tax documents, and other common form-based documents with decoy credentials, realistic names, addresses and logins, all information that is familiar to all users.

We note that the believable property of a decoy may be less important than other properties defined below since the attacker may have to open the decoy in order to decide whether the document is real or not. The act of opening the document may be all that we need to trap the insider, irrespective of the believability of its content. Hence, enticing an attacker to open a document, say one with a very interesting name, may be a more effective strategy to detect an inside attack than producing a decoy document with believable content.

Enticing: highly attractive and able to arouse hope or desire; "an alluring prospect"; lure.

Herein lies the issue of how does one measure the extent to which a decoy arouses desires, how well is it a lure? One obvious way is to create decoys containing information with monetary value, such as passwords or credit card numbers that have black market value [15, 26]. However, enticement depends upon the attacker's intent or preference. We define enticing documents in terms of the likelihood of an adversary's preference; enticing decoys are those decoys that are chosen with the same likelihood. More formally, for the document space M, let P be the set of documents of an adversary's A preference, where $P \subseteq M$. For some value ϵ such that $\epsilon > 1/|M|$, an enticing document is defined by the probability

$$\Pr[m \rightarrow M | m \in P] > \epsilon$$

where $m \rightarrow M$ denotes m is chosen from M. An enticing decoy is then defined for the set of decoys D, where $D \subseteq M$, such that

$$\Pr[m \rightarrow M | m \in P] = \Pr[d \rightarrow M | d \in D]$$

We posit that by defining several general categories of "things" that are of "attacker interest", one may compose decoys using terms or words that correspond to desires of the attacker that are overwhelmingly enticing. For example, if the attacker desires money, any document that mentions or describes information that provides access to money should be highly enticing. We believe we can measure frequently occurring (search) terms associated with major categories of interest (*e.g.*, words or terms drawn from finance, medical information, intellectual property) and use these as the constituent words in decoy documents. To measure the effectiveness of this generative strategy, it should be possible to execute content searches and count the number of times decoys appear in the top 10 list of displayed documents. This is a reasonable approach also, to measuring how conspicuous, defined below, the decoys become based upon the attacker's searches associated with their interest and intent.

Conspicuous: easily visible; easily or clearly visible; obvious to the eye or mind; Attracting attention.

A *conspicuous* decoy should be easily found or observed. Conspicuous is defined similar to enticing, but conspicuous documents are found because they are easily observed, whereas enticing documents are chosen because they are of interest to an attacker. For the document space M, let V be the set of documents defined by the minimum number of user actions required to enable their view. We use a subscript to denote the number of user actions required to view some set of documents. For example, documents that are in view at logon or on the desktop (requiring zero user actions) are labeled V_0, those requiring one user action are V_1, etc. We define a "view", V_i of a set of documents as a function of a number of user actions applied to a prior view, V_{i-1}, hence

$$V_i = \text{Action}(V_{i-1}) \text{ where } V_j \neq V_i, j < i$$

An "Action" may be any command or function that displays files and documents, such as 'ls', 'dir', 'search.' For some value ϵ such that $\epsilon > 0$, a conspicuous document, d, is defined by the probability

$$\prod_{i=0}^{n} \Pr[V_i] > \epsilon$$

where n is the minimum value where $d \in V_n$. Note if d is on the desktop, V_0, $\Pr[V_0] = 1$ (*i.e.*, the documents in full view are highly conspicuous).

Detectable; to discover or catch (a person) in the performance of some act: to detect someone cheating.

Decoys must ensure an alert is generated if they are exploited. Formally, this is defined for adversary A, document space M, and the set of decoys D such that $D \subseteq M$. We use $Alert_{A,d} = 1$ to denote an alert for $d \in D$. We say d is detectable with probability ϵ when

$$\Pr[d \rightarrow M : Alert_{A,d} = 1] \geq \epsilon$$

Ideally, ϵ should be 1.

We designed the decoy documents with several techniques to provide a good chance of detecting the malfeasance of an inside attack in real-time.

- At time of application start-up, the decoy document emits a beacon alert to a remote server.
- At the time of memory load, a host-sensor, such as an antivirus scanner, may detect embedded tokens placed in a clandestine location of the document file format.
- At the time of exfiltration, a NIDS such as Snort, or a stream event detection system such as Cayuga [5] may be used to detect these embedded tokens during the egress of the decoy document in network traffic where possible.
- At time of information exploitation and/or credential misuse, monitoring of decoy logins and other credentials embedded in the document content by external systems will generate an alert that is correlated with the decoy document in which the credential was placed.

This extensive set of monitors maximizes ϵ, forcing the attacker to expend considerable effort to avoid detection, and hopefully will serve as a deterrent to reduce internal malfeasance within organizations that deploy such a trap-based defense. In the proof-of-concept implementation reported in this paper, we focus our evaluation on the last item. We utilize monitors at our local IT systems, at Gmail and at an external bank.

Variability: The range of possible outcomes of a given situation; the quality of being subject to variation.

Attackers are humans with insider knowledge, even possibly with the knowledge that decoys are liberally spread throughout an enterprise. Their task is to identify the real documents from the potentially large cache of decoys. One important property of the set of decoys is that they are not easily identifiable due to some common invariant information they all share. A single search or test function would thus easily distinguish the real from the fake. The decoys thus must be highly varied. We define variable in terms of the likelihood of being able to decide the believability of a decoy given *any* known decoy. Formally, we define *perfectly variable* for document space M with the set of decoys D such that $D \subseteq M$ where

$$\Pr[d' \rightarrow D : \mathrm{Exp}^{believe}_{A,D,M,d'} = 1] = 1/2$$

Observe that under this definition an adversary may have access to *all* N previously generated decoys with the knowledge they are bogus, but still lack the ability to discern the N+1^{st}. From a statistical perspective, each decoy is independent and identically distributed. For the case that an adversary can determine the N+1^{st} decoy only after observing the N prior decoys, we define this as an *N-strong Variant.*

Clearly, a good decoy generator should produce an unbounded collection of enticing, conspicuous, but distinct and variable documents. They are distinct with respect to string content. If the same sentence appears in 100 decoys, one would not consider such decoys with repetitive information as highly variable; the common invariant sentence(s) can be used as a "signature" to find the decoys, rendering them distinguishable (and clearly, less enticing).

Non-interference: Something that does not hinder, obstructs, or impede.

Introducing decoys to an operational system has the potential to *interfere* with normal operations in multiple ways. Of primary concern is that decoys may pollute authentic data so that their legitimate usage becomes hindered by corruption or as a result of confusion by legitimate users (*i.e.*, they cannot differentiate real from fake). We define non-interference in terms of the likelihood of legitimate users successfully accessing normal documents after decoys are introduced. We use $\text{Access}_{U,m} = 1$ to denote the success of a legitimate user U accessing a normal document m. More formally, for some value ϵ, the document space M, $\forall m \in M$ we define

$$\Pr[Access_{U,m} = 1] \geq \epsilon$$

on a system without decoys. Non-interference is then defined for the set of decoys D such that $D \subseteq M$ and $\forall m \in M$ we have

$$\Pr[Access_{U,m} = 1] = \Pr[Access_{U,m} = 1|D]$$

Although we seek to create decoys to ensnare an inside attacker, a legitimate user whose data is the subject of an attacker must still be able to identify their own real documents from the planted decoys. The more enticing or believable a decoy document may be, the more likely it would be to lead the user to confuse it with a legitimate document they were looking for. Our goal is to increase believability, conspicuous, and enticingness while keeping interference low; ideally a decoy should be completely non-interfering. The challenge is to devise a simple and easy to use scheme for the user to easily differentiate their own documents, and thus a measure of interference is then possible as a by-product.

Differentiable: to mark or show a difference in; constitute a difference that distinguishes; to develop differential characteristics in; to cause differentiation of in the course of development.

It is important that decoys be "obvious" to the *legitimate user* to avoid interference, but "unobvious" to the insider stealing information. We define this in terms of an inverted believability experiment, in which the adversary is replaced by a legitimate user. We say a decoy is differentiable if the legitimate user always succeeds. Formally, we state this for the document space M with the set of decoys D such that $D \subseteq M$ where

$$\Pr[\text{Exp}^{believe}_{U,D,M} = 1] = 1$$

How might we easily differentiate a decoy for the legitimate user so that we maintain "non-interference" with the user's own actions and legitimate work? The remote thief who exfiltrates all of a user's files onto a remote hard drive may be perplexed by having hundreds of decoys amidst a few real documents; the thief should not be able to easily differentiate between the two cases. If we store a hundred decoys for each real document, the thief's task is daunting; they would need to test embedded information in the documents to decide what is real and what is not, which should complicate their end goals. For clarity, decoys should be easily *differentiable* to the legitimate user, but not to the attacker without significant effort. Thus, the use of "beacons" or other embedded content in the binary file format of a document, must be judiciously designed and deployed to avoid making decoys trivially differentiable for the attacker.

3.1 Decoy Document Design

The primary goal of the trap based defense is to detect malfeasance. Since no system is foolproof, we propose that multiple overlapping signals be embedded in the decoy documents to ensure *detectability*. Any alert generated by the multiple decoys is an indicator that some insider activity has occurred. Since the attacker may have varying levels of sophistication, a combination of traps are used in decoy documents to increase the likelihood one will succeed in generating an alert. A sophisticated attacker may, for example, disable the internal beacon, or cut off network connections avoiding communication, disable or kill local host monitoring processes, or they may exfiltrate documents via a web-browser without opening them locally. The documents are designed with several means of detecting their misuse:

- embedded honeytokens, computer login accounts created that provide no access to valuable resources, and that are monitored when (mis)used;
- embedded honeytoken banking login accounts specifically created and monitored for this trap-based technology demonstration specifically to entice financially motivated attackers;
- a network-level egress monitor that alerts whenever a marker, specially planted in the decoy document, is detected. Presently Snort may be used as simple signature detector as a proof-of-concept;

- a host-based monitor that alerts whenever a decoy document is "touched" in the file system such as a copy operation;
- an embedded "beacon" alerts a remote server. The web site emits an email to the registered user who created and downloaded the decoy document.

The implementation of features are described below.

3.1.1 Honeytokens

This layer of defense is made up of "bait" information such as online banking logins provided by a collaborating financial institution, credit card numbers, login accounts for online servers, and web based email accounts. The primary requirement for bait is that it be detectable when (mis)used. For example, one form of bait that we use are usernames and passwords for Gmail accounts. Our system is integrated with a variety of services to enable monitoring of these credentials once they are deployed as decoys. In the case of the Gmail accounts, custom scripts access *mail.google.com* to parse the bait account pages, gathering account activity information. The information includes the IP addresses for the previous 5 account accesses and the time. If there is any activity from IP addresses other than our system monitor, an alert is triggered with the time and IP of the offending host. Alerts are also triggered when the monitor cannot login to the bait account. In this case, we conclude that the account password was stolen (unless monitoring resumes) and the password changed unless other corroborating information (like a network outage) can be used to convince otherwise. In addition, some of our accounts have password monitors, allowing us to produce a seemingly unbounded collection of decoy variants for individual usernames.

In the case of financially motivated bait, we are beginning to use real credit card numbers in addition to banking login credentials. Many credit card providers offer "one-time-credit-card numbers" and other forms of Controlled Payment Numbers [19], which enable the generation of multiple credit card numbers for a single account. In the case of PayPal, single use credit card numbers can be generated with a predetermined balance. Our system monitor is being integrated with the PayPal APIs to automatically monitor the activity of the credit card numbers deployed through us. As is the case for all of the decoys, the benefit of deployment through our system is the automation, enabling their creation, monitoring, and distribution en masse.

3.1.2 Beacon Implementation

The highly sophisticated attacker will likely attempt to differentiate between a real document and a decoy by analyzing the binary file format prior to opening a file. This necessitates a design where beacon code and watermarks in decoy documents are hidden to avoid their easy identification. The attacker would surely avoid the decoys if they could easily identify them by a simple static test for an embedded beacon. The beacon code can be embedded in documents in a number of ways and made to appear statistically equivalent to its surrounding data using a blending technique called "spectrum shaping" (see [22, 6]). Such obfuscation techniques are very hard to defeat [16].

Using common techniques developed for malware, beacons attempt to silently contact a centralized server with a unique token embedded within the document at creation time. The token is used to identify the decoy and document, the IP address of the host accessing the decoy document. Depending on the particular document type and the rendering environment used during viewing of the beacon document, some additional data may be collected.

The first proof-of-concept beacons have been implemented in MS Word and PDF and deployed through our web site. In the case of the MS Word document beacons, the examples rely on a stealthily embedded remote image that is rendered when the document is opened. The request for the remote image is a positive indication the document has been opened. In the case of PDF document beacons, the signaling mechanism relies on the execution of Javascript within the document. Our web site will include a tutorial guiding the user on how to generate, download, and enable the decoys' silent communication on hosts. It is important to point out that there are methods for disabling the beacon mechanism.

3.1.3 Embedded Marker implementation

Beacon documents contain embedded markers that a host or network sensor may detect either when documents are loaded in memory or transmitted in the clear. The markers are constructed as a unique pattern of word tokens uniquely tied to the document creator. The sequence of word tokens is embedded within the beacon document's meta-data area or reformated as comments within the document format structure. Both locations are ideal

for embedding markers since most rendering programs ignore these parts of the document. The embedded markers can be used in Snort signatures or with a DLP such as OpenDLP for detecting exfiltration.

4 Host-based Sensors

A good decoy should make it difficult for an adversary to discern whether they are looking at an authentic document from a legitimate source or if they are looking at a decoy. For concreteness, we build upon the definition of "perfect secrecy" proposed in the cryptographic community and define a "perfect decoy" to be a decoy that is completely indistinguishable from one that is not. One approach we use in creating decoys relies on a document marking scheme in which all documents contain embedded markings such that decoys are tagged with HMACs (i.e., a keyed cryptographic hash function) and non-decoys are tagged with indistinguishable randomness. Here, the challenge of distinguishing decoys reduces to the problem of distinguishing between pseudorandom and random numbers, a task proven to be computationally infeasible under certain assumptions about the pseudorandom generation process. Hence, we claim these to be examples of perfect decoys and the only attacker capable of distinguishing them is one with the key, perhaps the highly privileged insider.

As a prototype perfect decoy implementation, we designed and built a component for adding HMAC markers into PDF documents. Markers are added automatically using the iText API, and inserted into the OCProperties section of the document. The OCProperties section was chosen because it can be modified on any PDF without impact on how the document is rendered, and without introduction of visual artifacts. The HMAC value itself is created using a vector of words extracted from the content of the PDF. The HMAC key is kept secret and managed by our system, where it is also associated with a particular registered host. Since the system depends on all documents being tagged, another component inserts random decoy markers in non-decoy documents, making them indistinguishable from decoys without knowledge of the secret key.

One of the key techniques employed by the architecture involves host-level monitoring of user-initiated events. The host sensor serves two functions. The sensor is designed to profile a baseline for the normal search behavior of a user. Subsequent monitoring for abnormal file search behaviors that exhibit large deviations from this baseline signal a potential insider attack. The host sensor also detects when decoy documents containing embedded markers are read, copied, or exfiltrated. The goal of the host-level decoy sensor is to detect these malicious actions accurately and with negligible performance overhead. Abnormal user search events that culminate in decoy document

access are a cause for concern. A challenge to the user, such as asking one of a number of personalized questions, may establish whether a masquerade attack is occurring.

Our prototype sensor has been built for the Windows XP platform and relies on hooks placed in the Windows ServiceTable. This is a typical approach used by malicious rootkits; however, in contrast to the traditional rootkit objective of remaining undetected, the host-level decoy sensor does not require operational secrecy. Our threat model assumes attackers have knowledge that a system is being monitored, but they must not know the identities of the decoys or the key used by the sensor to differentiate them. Furthermore, the attacker will likely not know the victim user's behavior, information that is not readily stolen such a credential or a key. Given that adversaries may be aware of system monitoring, special care must be taken to prevent the sensor from being subverted or, equally important, to detect if it is subverted. We have ongoing work aimed at preventing and detecting subversion of the sensor. One strategy involves a means to "monitor the monitor" to detect if the host sensor is disabled use of tamper-resistant soft- ware techniques. One possible solution we are investigating relies on "out-of-the-box" monitoring, in which a virtual machine-based architecture is used to conduct host-based mon- itoring outside of the host from within a virtual machine monitor. In an enterprise environment, integration with a DLP agent component (such as might already exist in the system) would offer an attractive alternative to using a custom-built sensor.

4.1 Detecting Perfectly Believable Decoys

The second host sensor also detects malicious activity by monitoring user actions directed at HMAC-embedded decoy documents. Any action directed toward a decoy is suggestive of malicious activity. When a decoy document is accessed by any application or process, the host sensor initiates a verification function. The verification function is responsible for differentiating between decoys and normal documents by computing a decoy HMAC (as described earlier) for the particular document in question and comparing it to the one embedded in the OCProperties section of the document. If there is a match, the document is deemed a decoy and an alert is triggered; otherwise, the document is deemed normal and no action is taken.

The host sensor performs tasks similar to antivirus programs. In evaluating the performance of the sensor, we would use overhead comparisons of

antivirus programs as a benchmark, since the task of comparing an HMAC code is not substantially different from testing for an embedded virus signature. Hence, accuracy performance is not revelant for this particular detector. However, there is a fundamental difference between the task of detecting malware and that of detecting decoy activity. Antivirus programs are designed to prevent the execution of and quarantine malicious software whenever any process is initiated. In decoy detection the objective is merely to trigger an alert when a decoy file is loaded into memory. Thus, the decoy detection need not serialize execution; for example, it may be executed asynchronously (and in parallel by running on multiple cores).

We have tested the prototype decoy host sensor on a Windows XP machine. A total of 108 decoy PDF documents were embedded in the local file system. Markers containing randomness in place of HMACs were embedded in another 2,000 normal PDF files on the local system. Any attempt to load a decoy file in memory was recorded by the sensor including content or metadata modification, as well as any attempt to print, zip, or unzip the file.

The sensor detects the loading of decoy files in memory with 100% accuracy by validating the HMAC value in the PDF files. However, as we discovered during our validation tests, decoy tests can be susceptible to non-negligible false positive rates. The problem encountered in our testing was created by antivirus scans of the filesystem! The file accesses of the scanning process that touched a large number of files, resulted in the generation of spurious decoy alerts. Although we are engineering a solution to this particular problem by ignoring automatic antivirus scans, our test does highlight the challenges faced by such monitoring systems. There are many applications on a system that access files indiscriminately for legitimate reasons. Care must be taken to ensure that only (illicit) human activity triggers alerts. As a future improvement to the sensor, file touches not triggered by user-initiated actions, but rather caused by routine processes, such as antivirus scan- ners or backup processes may be filtered. Nevertheless, this demonstrates a fundamental design challenge to architect a security system with potentially interfering competing monitors.

With regard to the resource consumption of the sensor, the components of the sensor used an average 20 KB of memory during our testing, a negligible amount. When performing tests such as the zipping or copying of 50 files, the file access time overhead averaged 1.3 sec on a series of 10 tests, using files with an average size of 33 KB. The additional access time introduced by the sensor is unnoticeable when opening or writing document files. Based

on these numbers, we assert that our system has a negligible performance impact to the system and user experience.

5 Search Behavior Modeling

The sensor collects low-level data from file accesses, windows registry accesses, dynamic library loading, and window access events. This allows the sensor to accurately capture data about specific system and user behavior over time. For example, we posit that one method to check if an insider has infiltrated the system is to model "search" behavior as a baseline for normal behavior. We conjecture that each user searches their own file system in a unique manner. They may use only a few specific system functions to find what they are looking for. Furthermore, it is unlikely a masquerader will have full knowledge of the victim user's file system and hence may search wider and deeper and in a less targeted manner than would the victim user. Hence, we believe search behavior is a viable indicator for detecting malicious intentions. Specific sections of the windows registry, specific DLLs, and specific programs on the system are involved with system searching applications. For a given time period (10 seconds in our initial experiments), we model all search actions of a user. After a baseline model is computed, the sensor switches to detection mode and alerts if the current search behavior deviates from the user's baseline model. Deviation measurements are made by examining a combination of the volume and velocity of system events in association with other user activities that should add some context to the user search actions, such as the number of processes being created and destroyed. Presently, this sensor component is being integrated in the architecture to function with the host sensor described next that detects decoy document accesses.

To evaluate this model, we first gathered user-event data to compute the baseline normal models, as well as data that simulates masquerade attacks. The dataset, known as the RUU dataset is described in Section 5.1, below. For the former, we had 34 computer science students install a host sensor on their personal computers. The sensor monitored all registry-based activity, process creation and destruction, window GUI access, and DLL libraries activity. The data gathered consisted of the process name and ID, the process path, the parent of the process, the type of process action (e.g., type of registry access, process creation, process destruction, etc.), the process command arguments, action flags (success/failure), and registry activity results. A timestamp was also recorded for each action. The collected data was automatically uploaded to a server, after the students had the chance to filter any data that they were not willing to share.

To obtain masquerade attack data, we conducted a user study in which 14 students had unlimited access to the same file system for 15 minutes each. None of the users had prior access to this file system, which was designed to look very realistic and to include potentially interesting patent applications, personally identifiable information, as well as account credentials stored in various files. The students were provided a scenario where they were asked to perform a specific task, which consisted of finding any data on the file system that could be used for financial gain.

The features used for modeling were in essence volumetric statistics characterizing search volume and velocity, and describing the overall computer session in terms of number of processes running, particulalrly the number of editing applications. A one-class Support Vector Machine (ocSVM) model was then trained for each user using those features. The same features were extracted from test data after dividing them into 10-second epochs. The ocSVM models were tested against these features, and a threshold was used to determine whether the user activity during the 10-second epochs was normal or abnormal. If the user activity was performed by the normal user, but was classified as abnormal by the ocSVM model, a false positive is recorded. Our results using the collected data and the modeling approach described above show that, we can detect all masquerader activity with 100% accuracy, with a false positive rate of 0.1%.

Extensive prior work on masquerade attack detection has focused on the Schonlau dataset for evaluation. The data set served as a common gold standard for researchers to conduct comparative evaluations of competing machine learning algorithms. The basic paradigm this work follows is a supervised training methodology where 5000 commands from each user serve as training data for the users normal behavior model. A classifier or model for each user is then tested against hold out data not used in training from the users command dataset but embedded in a random location with another randomly chosen users data. The performance results reported indicate the accuracy of the classifiers learned by a particular machine learning algorithm in identifying foreign commands, those blocks of commands deemed abnormal.

The model we chose to embed in the user search command sensor is different from these prior bag of command oriented models. Our current studies analyze user command events and the rates at which commands are issued using the RUU datasets described in the sidebar. Accuracy is estimated with respect to classification errors measured for each 10 second epoch of user

events. Furthermore, whereas the Schonalu data consists of Unix commands, the RUU datasets contain user events created in a Windows environment.

In order to compare our results with these prior studies, we need to translate the false positive rates in classifying blocks of 100 commands with the error rate of classifying user commands issued within each standard duration epoch. Unfortunately, the Schonalu datasets are devoid of times- tamps and a direct comparison of our modeling technique is not feasible. No one can accurately determine how long it takes each user in the Schonlau data to issue 100 commands. If we assume that it takes 20 seconds to issue one user command on average (a rough estimate from the RUU datasets for certain periods of time), our experiments show a detection rate of 100% can be achieved with a false positive rate of 1.4%. This is a 78% improvement in false positive rate over the best reported classifier in the prior Schonlau work. Indeed, none of the prior work reports a 100% detection rate at any reasonable false positive rate. If we assume it takes on average longer than 20 seconds to issue a user command, the results we achieved drops the false positive rate even further.

The comparison may not be entirely fair since the models and the data are quite different even though the data are generated by human users. The use of temporal statistical features from the RUU data set is crucial in modeling users behavior leading to far more accurate results than blocks of commands. Furthermore, in our work, we focus on user search events, limiting the amount of data analyzed and reducing the complexity of the learning task. The RUU datasets were created and are available to serve as a more up to date and modern gold standard for other researchers to perform similar studies.

5.1 Data and Evaluation

Research in insider attack is made difficult due to the lack of readily available insider attackers or a complete set of realistic data they generate. For this reason, researchers must resort to generating their own data that simulates insider attacks. The Schonlau dataset is the most widely used for academic study. It consists of sequences of 15,000 UNIX commands generated by 50 users with different job roles, but the data does not include command arguments or timestamps. The data has been used for comparative evaluations of different supervised machine learning algorithms. The Schonlau data is not a "true Masquerade" data set: the data gathered from different users were randomly mixed to simulate a masquerader attack, making the dataset perhaps

more suitable for "author identification" studies. An alternative approach to acquire sufficient data for evaluating monitoring and detection techniques is to devise a process to acquire human user data under normal operation as well as simulated attack data where "red team" users are tasked to behave as inside attackers. This type of study is typically subject to Institutional Review Board approvals since human subjects are involved. The process is costly, in time and effort but is sensible and appropriate to protect personally identifiable data of individual volunteer subjects. This was the approach taken by Maloof et al. for evaluating ELICIT. We as well gathered data from 34 users, all CS students at Columbia University, by distributing host sensors that upload system event data during normal system use. The population of student volunteers assures us the data they generate is derived from sources that have a common "role" in the organization, and hence variations in the user behavior and their data are not attributable to different job functions as is undoubtedly the case with the Schonlau dataset. We have also gathered data from 14 paid volunteers who emulated masquerade attacks on equipment provided in our lab. The dataset, which we call the RUU (Are You You?) data set, is over 8 GBytes and is available to legitimate researchers for download: `http://www1.cs.columbia.edu/ids/RUU/data/`. The data collected for each user averages about 5 days of normal system use, ranging in the extreme between 1 day and 59 days, and an average of more than 1 million records per user. Preliminary results using this data and the abnormal search benavior sensor described in the article show that the red team of masqueraders deviate substantially from ordinary system users.

6 Source Code Decoys

A software decoy system should have similar properties to those proposed for document-based decoys. Software decoys system should be inherently enticing. The decoy system must detect the exfiltration of bogus software that has been purposely planted in the system. To provide the means of detecting the exfiltration, the system we developed injects a beacon into the bogus software. In addition, the decoy system should be clearly conspicuous to adversaries. The bait, any bogus software, should be accessible and visible to adversaries and hence provided in a realistic honeypot. Lastly, the decoy system has a large set of bogus programs from original projects that are different from every other one. That is, variability should provide a decoy system with a variety of attractive bogus programs.

Code transformation is performed by inserting new code or modifying existing code into a seed program. There is a potential problem with such transformations. If only a small part of the program is affected by the transformation, and if the inserted or modified code doesn't look like normal code, it will be easy for adversaries and to identify bogus programs. Since bogus programs within the software decoys must be differentiated from original programs, the bogus programs themselves have additional core properties. They must be:

- *Compilable and Executable:* The bogus programs should be compilable without any error. The programs should be also executable for a reasonable amount of time so that the decoy can detect the software exfiltration and identify bogus software. The program that is to be successfully compiled should be run to produce observable behavior of the original software. These two properties are essential requirements to make the bogus software believable.

- *Indistinguishable:* An adversary should not be able to recognize whether a bogus program has been transformed from a particular source code or not. The adversary should have great difficulty in distinguishing bogus programs from a lot of other source codes.In other words, we should produce an unbounded collection of distinct and variable bogus programs. This property is crucial so that adversaries cannot easily determine whether a particular software source code is fake, nor that it is a derivative of an open source, non-proprietary project.

- *Believable:* The transformed program should logically look like a normal program. The program should have observable behavior similar to the original program. This property make adversaries trust it as if the bogus software were true and real source codes. While in the process of transforming on original seed program, we should try to maintain the original program structure and keep logical control flow so that the bogus software look likes real runnable source code. In addition, the logical structure and control flow can make bogus software resistant to some degree of static analysis.

Having discussed the properties that are need for decoy systems, and defined the additional properties that bogus bogus software should possess, we will show that these additional bogus software properties can be validated through extensive experiments with real open source projects. Widely accepted software metrics, such as similarity and software complexity, provide evidence of the practicality of our proposed bogus software generator.

6.1 Software Decoy Generator Architecture

This section will provide an overview of the system architecture that we designed and implemented to create a software-based decoy system, depicted in Figure 3. The system is given an original software project including several programs (in Java) as an input seed. It then produces a bogus project having a series of bogus programs. The generated bogus project is maintained in one of the software version control systems, containing different versions of the bogus project.

There are three requisite processes to create the software decoy: program analysis, code obfuscator, and program generator. First, the structure of each program in a project with diverse source codes should be syntactically analyzed. The program analysis output is an information table that has a data structure including the identifiers, type, scope, declarations, and relationships for variables and functions. Second, after the static program analysis, a code obfuscator transforms each of the original programs in a project into new bogus programs, using various code transformation methods that we designed. A fake project consists of a series of transformed bogus programs. Lastly, to generate different versions of the bogus project for software version control management, the generated bogus project is transformed during program generation. In addition, each of the generated bogus programs has a

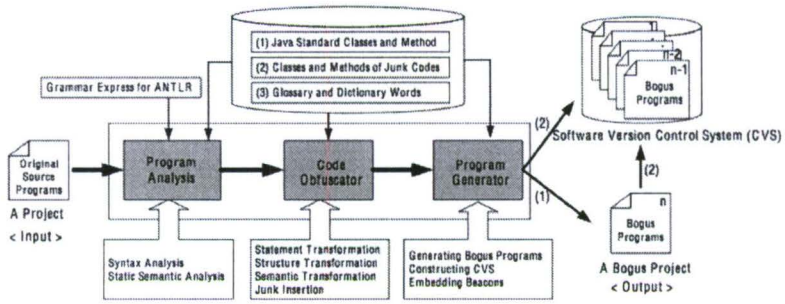


Figure 3: Software Decoy System Architecture

beacon inserted to be able to detect the exfiltration of proprietary software. The current target language is Java; however, the proposed method is generally applicable for all other languages such as C, C++, Python, C#, etc. The following subsections describe detailed methods for each process.

6.2 Analyzing Source Code

For any given input project seeding the synthesis of a software decoy, the proposed system first analyzes the syntax and the structure of each program in the project. ANTLR (Another Tool for Language Recognition) was used to extract information about syntax and the static semantics of each program. ANTLR is a parser generator with $LL(*)$ based on a context-free grammar argumented with syntactic and semantic predicates. The current prototype targets Java-based projects, but the proposed system is easily extended because ANTLR provides a flexible and language-agnostic grammar development environment.

As for program analysis of a target project, we obtain information about what classes/variables/methods are defined, how they are related and where they are used. The extracted information includes (1) package declaration and import information, (2) class and interface names, (3) member variables names and types, (4) member method names, types, and parameter information, (5) mapping between package and class/interface, (6) mapping between member variables and method, (7) mapping between class and interface, and (8) the scope of local variables.

The information is significant because when source codes are transformed

from one to another, other places matched with or related to the codes must be consistently modified in order to make the overall project compilable and runnable. In Java codes, symbols can be defined by package, classs, interfaces, variables, or methods. In addition, we also store the type information for the codes in order to avoid name conflicts. Local variables have different scopes depending on where they are defined. When code transformations are performed, the scope of related local variables must be determined to avoid errors where variables are undefined and uninitialized.

Finally, through program analysis, we create two databases that are used to generate a bogus project. First, we analyze the syntax of Java standard APIs to generate a database of standard classes and methods. This database is important to obfuscate a target program carefully since the APIs should be mostly preserved during code transformation. Second, we extract sample classes and methods from Java sample source codes collected from the Internet. This database is utilized to insert junk code in obfuscating target programs.

6.3 Obfuscating Source Code

After determining the syntax and the structure of target programs, the original programs in a project are obfuscated in order to generate a series of bogus programs. Code obfuscation has been used in a digital rights management system. The general obfuscation aims to make software as convoluted and hard to understand and to analyze as possible by automated reverse-engineering or by users. The obfuscated codes are often unintelligible and unclear on the surface of the code. Such code obfuscation techniques also enable us to create bogus programs that looks like real source codes, but are in fact fake. Such bogus programs make it difficult to distinguish between original source code and bogus source code.

The code obfuscator transforms original programs into bogus programs by making thorough changes in the form, syntax, or semantics of the original programs. This is called a code transformation as Definition 1. In fact, the code obfuscation in the literature preserves the semantics of a program. In other words, any transformation does not alter the program semantics, but rather it hides the semantics and makes them difficult to understand. However, the proposed system modifies the semantics of a program slightly while the program is being continuously transformed. The proposed system has four different code transformation methods: statement, structure,

semantic transformation, and junk code insertion. All the code transformation methods are closely related and the effects are interchangeably affected in programs since relevant variables/methods/classes should be changed together. In other words, the result of one code transformation might include the result of another code transformation.

1. *Statement Transformation:* This transformation renames all the variables and methods for each statement in a program. Based on syntax information from program analysis, it alters the name for all classes, methods, and variables in an original program. When changing all the names, the associated statements for variables, methods, and classes should be automatically renamed in all of the programs in a given project. The state- ment transformation replaces the original names for classes, methods and variables with bogus ones. When changing the names of classes and methods, the bogus names are selected in the database of glossary and dictionary words, as in Figure 3, according to user-defined themes, such as shopping-related, health-related, financial-related software, etc. The bogus names are redefined and renamed as appropriate words in the glossary database. This is a basic code transformation before applying other code transformations.

2. *Structure Transformation:* A program is structured in different lines in order to be more readable, but it does not have strict and firm rules. The original structure of a program can be changed in various ways: (1) reordering primitives and methods, (2) breaking abstractions, (3) expression change, (4) control structure modification, and (5) changing data types.

   First, we can randomize the placement of as many modules within a program, methods within a module, and statements within a method as possible. Second, by reconstructing new packages and modules, it breaks the original abstraction of a program, which thwart adversaries from understanding the original target program. Third, the proposed system replaces operators, such as assignment, multiplication, and comparison, into different expressions. There are an arbitrary number of ways to turn a given arithmetic expression into a sequence of different elementary statements. For example, multiplication by a constant is often turned into a sequence of less obvious adds and shifts. Fourth, the control structures in a program can be used interchangeably to alter the

structure of a program. The control structures include a conditional statement(e.g. if or else), a loop statement, (e.g. for, while), a selective statement(e.g. switch), and a jump statement(e.g. goto, continue, break). Lastly, data types in functions' parameters and variables are also changed if possible.

3. *Junk Code Insertion:* Bogus programs are diversified while generated in different ways by inserting any junk code as additional parts in a program. To insert junk code, there are several possible methods: (1) dead code insertion, (2) redundant statements, (3) method injection, and (4) code copy.

   First, the proposed system can add any number of blocks that can never be executed, such as classes, methods, etc. These are called dead code. Second, we place irrelevant or relevant statements for each line of a program. For instance, another variable or constant value can be declared and the variables are used any place in a program. Third, the proposed system clones bits and pieces of different methods in any given program, and the copied code looks different from the original one as a result of the code transformation, such as renaming, changing parameters in a method, etc. Lastly, from the database of classes and methods for junk code as in Figure 3, the proposed system selects one of them and reuses an arbitrarily chosen part of the code to generate bogus programs.

4. *Semantic Transformation:* We can also change the semantics of a program in different ways. First, the control flow of a program is naturally obfuscated while performing the proposed code transformations. Second, through call graph modification and data transformations, the semantics of an original program can be changed. Specifically, the use of inserted methods and inserted code blocks first tweak an original call graph. Second, the data transformation replaces data including constants and parameter values with other reasonable data. There are many ways to accomplish data transformation; return values can be changed or different constants are assigned. However, the transformed program should preserve the observable behavior defined earlier.

Definition 1 Let $T : P \rightarrow P'$ be transformation from program to program. T is code obfuscation, where P_B = T (P_0) has similar observable behavior.

T is a set of specific transformation elements, $t_1, t_2, ...t_n$. We enumerate several transformation techniques above. There are many other transformations possible, but what we have designed is sufficient for a proof of concept demonstration.

The bogus programs are also designed to have observable behavior that the adversary expects; in other words, the bogus program has the same or similar semantics as the original program. For example, in the case of software decoys designed for a bank, the adversary would naturally expect the program to have functions relating to withdrawal, deposit, interests, and so forth. We define these functions as observable behavior. We do not guarantee that the bogus program is correct or complete, but it does compile and appears to perform some "reasonable" functions. If we seed source code generator with an open source project, to increase the likelihood of producing a functional bogus program with believable and observable behavior, while avoiding disclosure of any functional information of the proprietary source code we aim to protect.

The generated bogus program, P_B, should be different from the original source program, P_0. The two programs can be evaluated according to two metrics: software similarity and containment. Similarity Δ is able to determine if two programs are very similar. Since the two programs, P_0 and P_B, should be very dissimilar, the similarity should be less than a threshold λ, per Eq. (1):

$$\Delta(P_0, P_B) < \lambda \tag{1}$$

Containment Θ evaluates if one program is partially contained in another. Because the transformed bogus program PB should have very small parts of code of the original source program Po, the containment should be less than a threshold β as in Eq. (2).

$$\Theta(P_0, P_B) = \frac{Numberoflinesmatchedbetweenbogussoftwareandoriginalsoftware}{Totalnumberoflinesinoriginalsoftware} < \beta \tag{2}$$

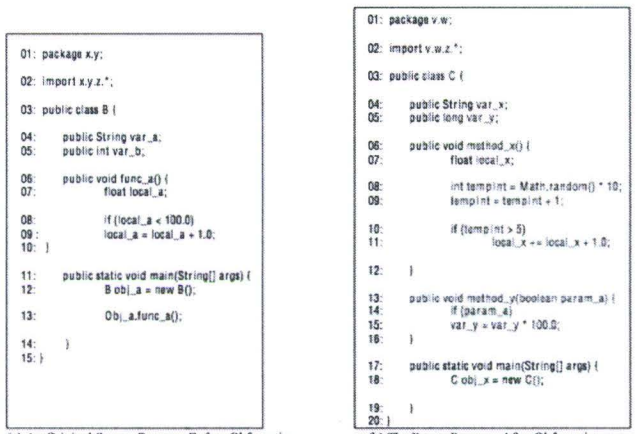
The Similarity Δ of two programs is a number between 0 and 1, such that when the similarity is close to 1, it is likely that the two programs will be approximately the same. Similarly, the containment Θ of P_B in P_0 is a number between 0 and 1 that, when close to 1, indicates that P_B is approximately contained within P_0. These two measurements are estimated by well-known software plagiarism tools.

As explained above, there are many different techniques for code transformation. Based on these code transformations, the proposed system can generate an arbitrary number of different bogus projects, as many as we demand. The current system stops generating a targeted bogus software when the similarity falls below a predefined thresh- old. This iterative algorithm makes it difficult for adversaries to detect any bogus software from real software, and can thus achieve one of the bogus software properties, indistinguishable.

However, each language has conventional rules which should be preserved when coding as well as transforming. First, Java standard libraries should not be changed, although we can add extra standard libraries to inject junk code on beacons. Second, keywords and reserved words should be preserved. There are many words, such as public, private, return, class, package, import, etc. These words are saved in the database as in Figure 3 to carefully transform source codes. However, as we stated, any user-defined names, such as package names, class names, method names, etc, can be changeable.

Some of the code transformation methods are often used for common optimization techniques. Copied codes, fake variables and functions are also used to protect software itself. In our method, these code transformation methods enable us to create as many bogus programs as we need. And if an adversary detects the use of these techniques, they may not be tipped of that the program is bogus. They may logically infer that these methods are being employed by the project to protect their software from reverse engineering.

Figure 4 shows an example of code transformation. As seen, the code obfuscation changed package names, class names, member variables, local variables, and method names. The relevant statements were also changed throughout all of the programs. We altered the data type "int" into "long" whenever feasible, as shown in line 5 of Figure 4 (b). In addition, we changed the expressions for the if-else conditional statement. Lastly, we inserted a junk method between line 13 and line 16, and injected multiple statements in lines 8 and 9, as shown in Figure 4 (b), that may be used in other places in the code.



```
01: package x.y;

02: import x.y.z.*;

03: public class B {

04:     public String var_a;
05:     public int var_b;

06:     public void func_a() {
07:             float local_a;

08:             if (local_a < 100.0)
09:             local_a = local_a + 1.0;
10: }

11:     public static void main(String[] args) {
12:             B obj_a = new B();

13:             Obj_a.func_a();

14:     }
15: }
```

(a) An Original Source Program Before Obfuscation

```
01: package v.w;

02: import v.w.z.*;

03: public class C {

04:     public String var_x;
05:     public long var_y;

06:     public void method_x() {
07:             float local_x;

08:             int tempInt = Math.random() * 10;
09:             tempInt = tempInt + 1;

10:             if (tempInt > 5)
11:                     local_x += local_x + 1.0;

12:     }

13:     public void method_y(boolean param_a) {
14:             if (param_a)
15:             var_y = var_y * 100.0;
16:     }

17:     public static void main(String[] args) {
18:             C obj_x = new C();

19:     }
20: }
```

(b) The Bogus Program After Obfuscation

Figure 4: Example Code Obfuscation for Software Decoy Generation

6.4 Generating Bogus Programs with Beacons encapsulated in a project

Based on the code transformation methods, the proposed system generates an arbitrary amount of decoy (bogus) software with any given input. The following outline below explains the method to generate a large number of different programs or diverse versions of similar programs. First, for any given input project, the proposed system generates different bogus software programs either from the original software or from the bogus software. Second, from the bogus software, the system produces a series of similar bogus programs so that software version control systems maintain a chain of history for the original project.

1. Generating different bogus software

   - From an original software: $P_0 \xrightarrow{T_j} P^k_{B_n}$
     (Note that $T_j = t_1, t_2, ..., t_n, i, j, k = 1, ..., n$ and t_i is a specific transformation in T)
   - From previous bogus software: $P^k_B \xrightarrow{T_j} P^l_{B_n}$
     Note that $Tj = t_1, t_2, ..., t_n and i, j, k, l = 1, ..., n$ and t_i is an element in T)

2. Generating various versions from the bogus software for the CVS repository

   $CVS(P^m_{B_n}) \xrightarrow{t_n} CVS(P^m_{B_{n-1}}) \xrightarrow{t_{n-1}} ...CVS(P^m_{B_i})... \xrightarrow{t_1} CVS(P^m_{B_1})$

   (Note that t_i is an element in T, m=l or k, and i,l,k = 1,...,n)

Looking at the first step in more detail, the proposed system creates a variety of decoy software from an original source code. Each resulting bogus software is different from every other one. In addition, the system uses previous bogus software to generate other new and different bogus software. The resulting bogus programs are dissimilar to each other depending on the number of iterations of code obfuscation (T). For any given input, the code transformation produces different kinds of decoy programs that are less than a predefined threshold of similarity.

Second, a project is managed by software version control systems, such as subversion, git, etc., to keep updating new codes and tracking different software versions. To make decoy software realistic, the bogus software should be maintained to look like a real project by using one of the software version control systems. We generate a series of different versions from the first resulting bogus software under the CVS version control system.

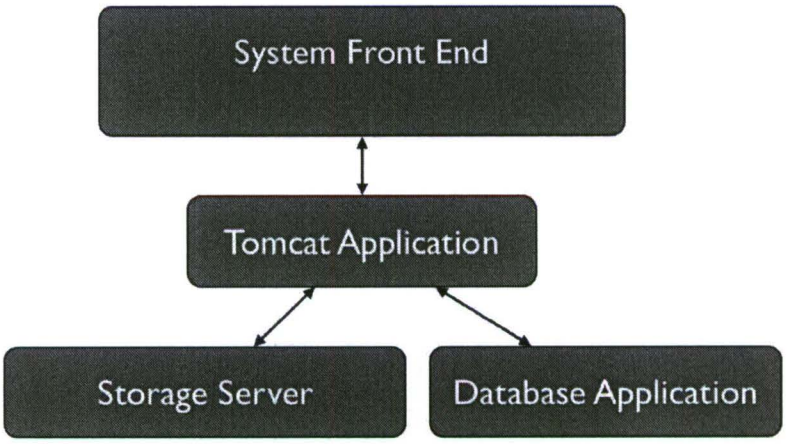
Specifically, the code transformation(T) has a set of different elements, t1, t2, ..., tn. One element of the transformation method, ti, is selected to generate slightly different versions of the bogus program every time. The different versions do not need to be dramatically transformed, and each version in a series of bogus software from one particular input is analogous to every other version. For example, it would be sufficient to generate different versions from a bogus program by simply changing one single statement, such as altering one variable declaration. However, to make it more realistic, the proposed system tries to maintain approximately the same number and modification size of two consecutive versions as the original project. The statistics used to generate an archive with typical number of updates and modifications were gleaned from a sample of Open Source Project Archives. Thus, the generated archive should appear as a realistic project.

Finally, each bogus program has a stealthy beacon that provides a signal indicating when and where the particular bogus program was used. The beacon plays a valuable role in identifying the exfiltration of software, by throwing an alert to a server. It is related to detectability of the decoy. In software decoys, the beacon can be embedded in several ways: in documents

accompanying the source code, at the time of compilation, at the time of execution. Typically, software provides several documents such as guidelines for compiling and execution instructions, and an API description. For those documents, our proposed system adopts a technique to embed the beacon into PDF or HTML. The signal mechanism utilizes the execution of Javascript within the document.

For software embedded beacons, we can embed the codes that sends signals to a server upon program compilation or execution. Several techniques can be provided. First, the bogus program can be modified to use a library that must be downloaded in order to successfully compile it. Then, the request for the library on the server is a positive indication the bogus program is about to be compiled. In addition, similar activities can be performed when the bogus program is about to be executed. For example, when the library is first loaded into memory, the library initialization routine should be able to play this role by using library constructor and destructor functions for dynamically loaded (DL) libraries.

Figure 5: Server Architecture



7 Application Server

By design, the allure defender system is language and architecture independent. Figure 5 shows a forward facing system, application host, and backend support modules.

Our initial prototype was deployed on a tomcat application server due to the specific Java libraries we are utilizing. These java libraries and packages can create, manipulate, and monitor a wide variety of document formats including pdf, microsoft word, and excel.

Tomcat [27] is a java based application framework which can be deployed on most modern operating systems and works in conjunction with java servlet technology to seamlessly bridge backend servers with front end website.

The components will communicate using the defined API and be packaged as independent components to allow maximum flexibility with the current system, and the ability to extend the system using standard programming components.

7.1 Requirements

Tomcat requires an underlying operating system and host system to run. We currently are running a test system as a virtual machine on a Ubuntu Linux host vm. The tomcat version is the one packaged with the current version of Ubuntu 11.04 (As of July 2011).

7.2 Scale-ability

In order to allow scale ability of the system we have modulated specific components and standardized a common api between components. In addition, we have designed it to be fail-safe, so that no specific module's failure will render the system inoperable. For example, the database hook class, allows the database backend to fail-over to a cache file, so that when it comes up again, we can forward the cache requests to the database.

7.3 Customization

System customization can be passed as arguments during start-up or specified in specific table settings. There is a configuration section which defines the following initial system customization:

- Title system title which shows up on the correspondent.
- Version short version string to show when deployed and what version of the software it is running
- DBname database name.
- DBhost database host machine.
- DBUser initial database user for setup.
- INFOLINK url to more information about the local system (if external).
- LIBlocation libraries needed by the system if external.
- URLLink main website link.

This list can be extended and customized. A GUI windows allows these parameters to be changed, and the values are kept in a text file which can be edited. These values are accessed during system run-time and can be configured for each installation.

7.4 Administration

Tomcat includes a management webpage which allows one to deploy, reload, undeploy component packages. It uses ".war" files which are jar like files containing the resources necessary by the application being deployed. In addition, command line programs bundled with tomcat allow administration to take place.

7.5 Upgrading

Deploying an updated version of the component is as simple as either copying an update ".war" file into the WEBAPPS tomcat directory or undeploying and redeploying the application via the tomcat admin page.

7.6 Debugging Problems

Tomcat is configured to dump it standard output and error output to a file called "catalina.out" in whatever the default run directory will be. When there are problems, simply check the latest catalina.out file for information. In addition the system logs internal actions to the logger subsystem which are kept in the log tables in the database.

7.7 Past Known issues

In older versions of tomcat it was necessary to pass a java headless argument to the underlying engine hosting the application. This has been fixed in newer versions. In addition older versions of tomcat has been too strict on what applications were allowed to execute, which was problematic since our application processes and stores documents on the local system on behalf of the user. These issues do not exist on the last few versions of tomcat.

For security reasons, it is best if the system is run as a non administrator user, so in case security is compromised, the damage will be minimized.

8 System Front End

The system front end represents the forward facing end of the system which allows the user to

- Manage accounts
- See an overview of decoy activity
- View system logs
- Customize the installed system.
- Backup and restore
- Database maintainance

8.1 Web-based GUI

The front end of the Allure defender will be a Java servlet website running on Tomcat on a Linux platform. The document generation plugins will be developed in Java in addition to the backend system. The system will allow the administrator full control and customization of the running allure system.

8.2 Accounting system

The accounting system maps users, roles, account status, emails, passwords, groups and other information related to accounts on the local installed system.

8.2.1 Account Creation and Registration

Accounts are created in one of two ways. Users can request accounts using the registration button on the main webpage (when enabled). An email is required so that a verification code can be emailed to the user to allow an email to be verified. if account passthrough is enabled, once verified the user account is created with lowest level privileges. If account passthrough is not enabled, the administrator is notified that an account is waiting approval. Each user can belong to a specific group (default group otherwise), which allows the managers to subdivide users by group.

Another option is for an administrator to import accounts en mass using a tab-delimited text file or similar means. Roles can be specific in the file or one role can be applied to all imported users.

A final option, left for the future, is to integrate with an organization's LDAP server.

8.2.2 Account Roles

Roles represent the user privileges in the system with the following roles specified:

- **Root**: these users can customize the local install and restart the system as necessary. They can opt to receive specific alerts upon exceptional predefined events. Root level users are expected to be installers. In addition Root users are also Administrators.
- **Administrators**: these users can add other users and view high level reports on all groups users which they are associated with. There is a 1 to N mapping between Administrators and groups. They can enable/disable accounts. Reset passwords and also change user level basic information (e.g., email, group).
- **Managers**: these users can view high level reports on specific groups of users. There is a 1-to-N mapping between managers and user/groups. Managers can set user level permissions and enable/disable accounts. They can also reset passwords.
- **Users**: these are specific users which belong to specific groups. This role allows a user to create or checkout decoy system units (e.g., documents) and/or manipulate template, etc.

8.2.3 Account Status

Each account in the system will be associated with one of the following states:

- **enabled**: active user with specific privileges associated with the account
- **waiting**: approval: recently created and waiting confirmation
- **disabled**: account has been disabled, this includes expired accounts

- **unconfirmed**: unknown status default if not set specifically in the system.

8.3 Setting Customization

System setting will be customized using either a text editor on specific text files in a config directory or using a front end gui to view and choose specific customizations. The values will be loaded at startup into the database so that can also be updated via a plaint text file placed in a specific loading directory.

The system will provide reasonable default settings for all resource settings, text strings, local settings to allow an installation to work with minimum user customization.

Specific customization settings of the system will be able to be saved, loaded and compared so that administrators can trace customization updates over time for debugging and forensic purposes.

As an example, one should be able to enable or disable the registration button from appearing on the main webpage. This will have the effect that only administrators can add accounts, rather than having users request accounts. Which window are displayed, the titles, the types of decoys users can create will all be controlled by the customization system.

9 Database Backend

The database ties together the components of the system to allow users, decoys, documents, and activity triggers to operate smoothly. A relational database stores local user accounts, generated documents, content information, use logs, decoy logs, and system settings.

The database does not necessarily have to be deployed on the same server as the deployment host. This will allow for security, redundancy, and scalability. In general we try to mitigate against any single point of failure of the system. In order to generalize the system to database communication, a database communication class is defined per database type. The purpose is to present standard method calls to the system.

This design allows the main system to use standard method calls, which are translated by each database communication class into low level sql calls.

As an example, our initial implementation implements a MySQL database hook and java based DB hook. The advantage of the java based db is that no other software needs to be installed or configured.

The system defines database interfaces to deal with specific database types and database manager which is implemented to create/adjust specific database tables. In general these operations only occur during setup and backup and rollback and reloading of tables. The actual communication is done with the database handle classes.

Interface DBConfig - set of functions which will be needed when setting up the db. Each table should be a separate method call so that new tables can be defined and modified in the future.

Class DBSetup - this class will be responsible for calling setup functions which the individual database hook classes will support. They define the schemas per table, which each database type might implement slightly differently.

Class JavaDBHook - will interface to derby based db installations. When the system uses a Java based db this is the class which talks to the software.

Class MysqlDBHook - will interface to mysql installation. This require a prior mysql installation either local or remote with correct user level permissions to create and modify tables in the database.

The following general database functions will be supported by each individual class:

- void AddUser(String Email,String IP,String Password,Int GroupID) - Allows the system to add a user to the accounting system. A user is associated with an email address and some ip for accounting purposes. Passwords are kept as MD5 strings in the database. GroupID is unique number associated with this user and another table which defines the permissions this user has.
- int CheckUserStatus(String Email) Checks the user status if enables or disabled or expired or suspended.
- SetUserStatus(String Email,int status) Checks the user status if enables or not.
- boolean CheckUserPassword(String Email,String MD5Pasword) checks if the password for this user matches
- boolean RequestChangeUserPassword(String Email) set in motion process to change the user process
- boolean ChangeUserPassword(String Email,String MD5Pasword) change the password for this user
- boolean AddBeaconToken(String Email,String beacon) registers the beacon to this user
- boolean CheckBeacon(String Beacon) do a check and send out an alert based on user preferences.
- HashMap GetUserPreferences(String Email) get all user preferences in the system
- boolean SetUserPreferences(String email,HashMap prefs) set the users preferences
- int GetUID(String Email) get users id in the system
- int GetGUID(String Email) get group id in the system
- void SetRelated(UID,GUID) add user to the group
- Hashmap CheckUserAlertPreferences(int UID) check the users alert choices

- boolean BackupDatabase(dbname, location) set in motion a snapshot
- boolean RestoreDatabase(dname, location) set in motion restore
- String GetVersion() get back information about current deployment
- boolean SetVersion(String) set local version information
- void SetLogEvent(int type, String body, int sourceflag, String IP) log some unique event

Again, each hook class, translated these high level calls to the specific sql/schema mapping.

9.1 Failover Redundancy

The database is the heart of the backend of the sytem. As such, this represents a critical layer of failure. To protect the integrity of the system we recommend a master slave database setup with automatic sync. The setup if beyond the scope of this manual.

From a high level, events in the datasbase are atomic (i.e. do not represent specific states) which makes it easy to sync deltas between master slave databases or backup the database every time X.

9.2 Tools

The frontend of the system depends on the database functioning correctly. We have included various tools to allow easier maintaince of the database backend. With more sophisticated database installations, any outside tools can also be utilized to administrate the backend database.

9.3 Maintenance tools for DB

The database communication classes need to allow the sytem to backup and reload a snapshot of the database. Specific routines are called from the gui facing functions in each database hook class to allow this to take place.

Table 1: User Table

Name	Type	Default	Note
Username	varchar(128)	not null	Name associated with the account (if any)
Email	varchar(128)	not null	Registration email
CreationDate	sqldate	current	Date account created
CreationIP	char(26)	not null	Where verification request came from
Status	short int	unknown	Account status
RegistratonDate	sqldate	current	
RegistrationIP	char(26)	not null	
UserID	unique long	not null	Unique ID of user internal in the system
PasswordHash	char(64)	not null	

Table 2: Group Table

Name	Type	Default	Note
Group	varchar(64)	not null	String of group
GroupID	Unique long	not null	unique long id of group
StartDate	sqldate	not null	When group was created

9.4 Table Schemas

We now outline all the internal tables in the database.

Table 3: User Group Table

Name	Type	Default	Note
UserID	long	not null	User ID
GroupID	long	not null	Group ID
Date	sqldate	not null	When user added to the group

Table 4: Account Creation Table

Name	Type	Default	Note
Name	varchar(128)	not null	Registration Name
IP	char(26)	unknown	Registration IP
SecretHash	char(64)	not null	Verification string
Date	sqldate	current date	When requested
Email	varchar(128)	not null	Email used

Table 5: System Settings Table

Name	Type	Default	Note
MainTitle	varchar(255)	BLANK	Title on main page
DataBaseType	varchar(32)	JavaDB	Which database is being accessed
DBName	varchar(64)	AllureDB	Database name
DBPassword	varchar(64)	AlLuReDb	Password to database
DBHost	varchar(255)	localhost	Host of the database
DBUser	varchar(255)	root	User to access the db
defaultEmail	varchar(255)	rootlocalhost	Main admin email

Table 6: Alerts Table

Name	Type	Default	Note
AlertCode			Unique ID for the specific code condition
ConditionAlert			What condition is met for the alert to be triggered by the system
Action			What action to take upon the alert condition being met
Status			If enabled or disabled

Table 7: Notification Table

Name	Type	Default	Note
AlertCode	int	not null	Which alert level to send on
UserID			User ID associated with this alert
Subject			Custom subject line for alert
Body			Customized body message for the alert
AlertString	varchar(255)		plain text of alert
CheckCondition	varchar(255)		some kind of condition to match
CheckEquality	int		if less than great or equal to condition
CheckValueThreshold	varchar(32)		some value to alert on

Table 8: System Log Table

Name	Type	Default	Note
IP	char(26)		What ip is associted with this log
Date	sqldate		Date of message
Activity	varchar(255)		predefined string of activity status
Location			local system or remote

Table 9: Usage Log Table

Name	Type	Default	Note
Date	sqldate	Current	Date of usage
IP	char(26)	NOT NULL	IP asociated with usage
UserID	int	NOT NULL	
ActivityCode	NOT NULL		
Note	varchar(255)		More information about the specific usage of the system

Table 10: HMAC Table

Name	Type	Default	Note
UID	int	NOT NULL	User in the system
Pattern	String	NOT NULL	Regular Expression Seed
Flag	int		Good vs Bad for HMAC
Date	int		When this hmac created

Table 11: Decoy Creation Table

Name	Type	Default	Note
Date	sqldate		When Beacon requested
IP	char(26)		IP requesting it
UserID	int		User associated with the beacon
BeaconHash	varchar(255)		Hash Represneting the beacon
BeaconType			Code for which type of beacon this is
FileLocation	varchar(255)		file created and its location
ContentInfo	blob		variables which were used during the creation process
TemplateInfo	blob		Seeds used to create the document.
DocumentID	varchar(255)		unique document id

Table 12: Decoy Activity Table

Name	Type	Default	Note
Date	sqldate		
IP	char(26)		
BeaconHash	varchar(255)		unique hash
Note	varchar(255)		
UserID	int		user associated with the beacon
BeaconType			

Table 13: Alert Configuration

Name	Type	Default	Note
UserID	int		User ID from account tables
Destination	varchar(255)		Where the alert is going
Type	int		Type of destination (email, text, etc)
Enabled	boolean		is alerts enabled
Note	varchar(255)		Note associated with alert

Table 14: Alert CC Table

Name	Type	Default	Note
UserID	int		User ID
CCUserID	int		Which ID should be ccd on the alerts
BeaconHash	varchar(64)	blank	Specific hash to CC (blank means wildcard)
Note	varchar(255)		Note associated with alert

10 Logging infrastructure

The aim of the logging system is to create a trail for forensic and accountability of the system. User level activity such as account creation, tweaking, log-in/logout, etc will be stored with a timestamp and when possible associated network ip associated with the action. In addition the system startup, reboot, and exception events will be logged to the central logging system.

```
Interface Logging {
boolean WriteLog(int logtype, String title, String Body, int flag);

ArrayList GetLogs (int logtype, int flags, Date StartDate, Date
EndDate, boolean matchregexp, String regexp);

boolean exportLogs (int logtype, int flags, Date StartDate, Date
EndDate, boolean matchregexp, String regexp, String filelocation,
boolean overwrite, char delimin);

boolean importLogs(char delim, String filelocation, boolean
eraseFirst);

int getLogSize();

String getLogStatus();
}
```

10.1 System Logging

System level logging will log start/shutdown. Exception events, and system interaction between component. Basically when database entries are written we would like to see some sort of activity in the system log table to be able to trace issues if something goes wrong.

10.2 Usage Logging

User level activity such as document creation, sign on and off, document retrieval. template usages, etc.

10.3 Decoy Logging

Decoy activity such as creation and pings are logged here. Any information which is based in as part of a beacon is also associated with these records.

The front end will also allow the system user the ability to search through the logs by log type, constrained by time, using some keyword matched. The results of the entire logs, should be available for export, depending on user privilege level.

10.4 Backup and Restore

This section of the system will allow the administrator to create a snapshot of the current files, settings, and decoy objects to backup a point in time. In addition, the backend database should be able to be backed up for both maintenance and forensic purposes. We will dump the raw sql to a tar.gz file with database table setup dumped separately (rather than inline). The system will create metadata so that points in time can be named so that rollback to a specific state can occur. This will be beneficial for testing out new features and plugins.

11 Decoy System

The decoy system consists of enticing beacons embedded within some container entity. These beacons when triggered will alert the system and can be associated with a specific user, time, machine, and event. From a user point of view the beacons can be passive beacons (no user notification, such as file monitoring, or quiet dns lookup), half passive (request permission, pdf remote url request, remote video), or active (need user permission, prompt to continue). Users can create specific decoy containers for deploying specific decoy types. Some container object might contain multiple beacons in order to increase the likely hood that one will return and trigger a real time alert.

11.1 Decoy Mechanisms

In order to facilitate easy decoy creation, the internal beacon class exposes the following interface:

```
interface DecoyObject {

String GenerateToken();

void CreateBeacon(Unique StringHash);

String GetDecoyObject();

}
```

- GenerateToken generates a unique hash string
- CreateBeacon takes a unique token and creates an internal beacon representation
- GetDecoyObject returns a text based representation of the decoy which will embedded by the particular container holding the beacon

11.2 Beacon Notification System

A beacon is considered to ping home if a unique string associated with any of the embedded beacon is passed back to some system and triggers a match

against an internal database and fires off a notification in the system. Since beacons are created to be unique, once received they need to be matched in the beacon registration system to a specific user. if a beacon string does not match, we must assume that something or someone is tampering with a the beacon document. Since by the design there should be a zero likelihood of anyone stumbling upon the specific beacon phone hone string. In fact, detecting the non matching beacons, is itself is an activity which can generate an alert etc.

One can leverage CGI GET request to send back an arbitrary length string with multiple command arguments to a specific server listening and correctly processing the requests. This can come in over open or encrypted channels since the random strings do not have any specific significance outside of a particular installation.

We now describe implemented technologies which can be used alone or in-conjunction with each other in a document container to signal an activity beacon alert.

11.2.1 Remote image references

A remote image is fetched from within a document. In many documents, one can embed a url so that the image is fetch and displayed inline. The beacon system process remote image fetches and will return a valid image, while at the same time register the beacon signal with the system and pass back any information sent home with the beacon.

The user can assign a specific image to be returned or the system can return a blank image (1 by 1 pixel of white square).

11.2.2 Tiny URL

A tiny url is created from a beacon url and embedded in the document. The idea is that we can partially annoymize the url fetch in cases where the user is prompted for confirmation. Tinyurl's are a standard way of sharing complicated url information.

11.2.3 Remote URL touches

When we can embed a remote url, for example as part of a video movie or a stand lone web document, we can monitor for these urls and either serve legitimate content, or simply server an HTTP 404 (Not Found) error. One

good example is a robots.txt file which the system would be expected to see as request (the fact that it returns a 404 should help the beacon be more believable.

11.2.4 SIP phone number

When we can embed a phone number, in conjunction with a VoIP PBX or other device that we can control or monitor, we can detect access to these phone numbers and/or extension. While monitoring can also be done in the organization's PBX (looking for outgoing calls to the decoy number/extension), this can be evaded by an adversary that uses another device (e.g., a cellphone).

11.2.5 DNS

When we can use realistic but unique DNS names whose resolver servers we can monitor (either through logs or through network eavesdropping), we can identify accesses or even reconnaissance activity by an adversary, since a requirement to access a system is to determine its IP address based on its DNS name. In some cases (e.g., over-eager browsers and similar components that do either prefetching or other types of network optimization), such queries may be emitted even without the adversary having to undertake specific action. In that case, these DNS names will act as beacons.

11.2.6 Honeypot and server log in information

In general, the enticing information (EI) embedded in the decoy documents may contain URLs for different types of protocols, pointing either to honeypots or to legitimate servers. Generally, the EI will contain a uniquely identifiable component (e.g., a file path or username) that will allow us to differentiate among decoys that point to the same general resources (honeypot or legitimate server). In the case of using legitimate servers, the use of fake username/password information (which will be rejected but logged by the server) in conjunction with log monitoring can provide a highly believable and practically inexhaustible source of EI.

11.2.7 Monitored Credit Cards

Previous work with a large banking company allowed us to use valid to empty visa accounts to track the proliferation of online fraud in relation to decoy documents. With the appropriate relationships to a financial institution (e.g., a bank, PayPal, Google Wallet, etc.), this can be extended and scaled up.

12 Document Generation System

The document generation system will combine the decoy system, content generation, and output formatter to allow the system to create enticing decoy documents. the template system will adopt content sets to customize the generated documents.

12.1 Architecture Overview

12.1.1 Decoy System

Will generate a unique decoy string which will be used by decoy object to generate decoy beacons. The method S"tring getUniqueDecoyString()" will generate a unique decoy string based on random number and current time and number of objects in the db. It will be then hashed to the create a unique md5 of the decoy string. This will be used as input for a specific decoy object.

12.1.2 Document Template Module

Document templates are composed of a tree like structure of the components of a document, and a set of xml like form objects which are embedded in the document structure and give rise to the content of the document.

A Template object is a root node and list of sub children node. Each node has a layout object which specifies which part of the document (if any) it will lay itself out. We will use the gridlayout scheme to specify object layouts on a paper document. in this way a template object can be visualized using some gui frontend which will allow the user to customize and name each template instance. We are modeling this after a mailmerge program where one can outline a document and mark placeholder for specific values to be replace in the document. Template replacement include 1 to 1 and n to 1 (either cycled or randomly chosen). the template content will be provided at runtime with the attached generation objects.

For example, an Email Message Decoy would be a composed of an Email Template Instances describing an email document layout. And will reference replacement fields from a Sender, recipient, and conversation objects. The conversation object might be a login conversation or vacation delegation conversation depending on the generation system choice at runtime.

12.1.3 Template Variations

- Medical Forms
- Setup information for system
- Tax Records
- Shopping List
- Credit card application
- Bank Statement
- Mortgage application
- Online shopping receipt
- Cell Phone bill
- Account setup information
- Printed Email
- Scientific report
- employment (hiring/firing) letter
- resume

12.2 Content Generation From Templates

The template content generation will be achieved using a collection of conversation objects, Identity Objects, and an associated template types. A conversation object requires one or more identity objects. The conversation object, has a collection of forms which it queries the attached identity objects for specific information. That way sentence structures can be tailored to correct male/female and topics etc.

- UserInfoClass - Object representing an individual user. Pertinent information includes date of birth, name, Address, Status, Income, Social

- CorporationObject - object representing some kind of corporate entity which will be used in some of the conversation information representations.
- Conversation - Object representing some snippet of information flow between either set of users or user and corporate entity. Document Output Module

The output module will take a template output and pass it to the document generation to create a specific pdf/doc/rtf/etc format. Once successful the output module will also register the embedded decoy string and associate user with the database so that when beacons are received the user can be notified with the specific beacon. In addition a log with a string sequence representing the choices from each module piece (template, conversation, identities) will be logged so that the document can be regenerated if the system needs to upgrade or update any parts of the documents creation process.

```
boolean MainDocumentCreateAndRegister (
DBHandle DatebaseHandle,
String FileSavePath,
String DocNAME,
Logger LogHandle,
DecoyObject BeaconDecoy,
boolean verbose\_setting,
DocumentTemplate docSource,
Conversation createdConversation,
UserInfoClass person[ ]
String UniqueToken,
String UserEmail );
```

13 API

In order to allow the system to be useful, there is a web api framework which allows users to fetch and generate decoys without having to sign in through the web frontend. This allows system to extend the Allure defender functionality using their own front end or gui to interact with the underlying architecture.

The web based api relies on ssl socket being enabled in the tomcat server. This gives the user the ability to pass in user credentials and specific requests.

In addition, Beacon generation and document generation have been packaged in indivdual jar files to allow easier deployment to stand along system.

13.1 Single Requests

A secure URL with the following parameters needs to be called in order to generate a single document return.

```
\url{https://installurl:port/srequest?user=U&md5pass=M&single=1&doctype=D&tempate=
```

where

- installurl is the installation of the system
- port is the port it is listening for ssl connections
- U is the user in the system
- M is the md5 of the password
- D is the document type as defined in the system (1 pdf, 2 word, etc)
- N which template is used (this depends on the system and user settings).
- T is the content variation type (email, reciept, etc).

The return is a binary file with the content requested.

13.2 Batch Documents Creation

A secure URL with the following parameters needs to be called in order to generate a single document return.

```
\url{https://installurl:port/mrequest?user=U&md5pass=M&single=1&doctype=D&tempate=
```

where

- installurl is the installation of the system
- port is the port it is listening for ssl connections
- U is the user in the system
- M is the md5 of the password
- D is the document type as defined in the system (1 pdf, 2 word, etc)
- N which template is used (this depends on the system and user settings).
- T is the content variation type (email, reciept, etc).
- H is a string representing some mix of documents
- Z is the number of

The return is a binary zip file with the content requested inside of it.

13.3 Future Additions

Future additions will include ability to send up and mark documents and the ability to request enticing content and enticing decoys for use in outside applications such as blogs, instant messages, etc.

The api can be also moved to secure sockets, to allow custom tools to query and pull decoy information.

14 Deployment Guide

This section described how to setup a new deployment of the Allure system.

14.1 Obtaining the latest software

The latest copy of the software should be available in the svn repository. Currently on fog.cs.columbia.edu. The project should have a live .war file or can be rebuilt from source. There should be a build.sh script which will allow the user to rebuild the system on their local os. Since this is java, the latest .war file should be enough along with some specific support files to be able to run the system

In addition there are a set of files include in bundle.zip which need to be unpacked in the runtime directory of the running system. This is different depending on how tomcat is deployed. For simplicity, use the Allure front end to add the specific files to the system.

14.2 Installation Requirements

The system needs a basic Operating System and a local Java runtime package to deploy. One can also make use of specific database backend such as mysql. The deployment and installation of mysql is beyond the scope of this document. By default, there is a java based database which is included in the main .war file.

In addition, Tomcat is required for running the .war file. Tomcat can be installed from OS packaging or as a standalone installation. Specific security settings on tomcat should be investigated, to lock down the installation, but details are beyond the scope of the document.

14.3 SSL Support

To allow a secure channel browsing experience, one needs to turn on SSL support on Tomcat. The instructions are on Tomcat's setup guide. Allure Defender is currently agnostic as to whether SSL is being used, although this should become a requirement for a final product and for remote batch processing.

14.4 Sample Install

- If you are starting from scratch, install and update your favorite OS.
- Install Java SDK (JDK if you are building from source)
- Install Tomcat. It would be advisable to create a non admin user and download and unzip the tomcat files as this user. This allows tomcat to run, but in case of compromise, will not affect the entire system (one hopes).
- Setup SSL at this point. Also setup tomcat admin account and any permission files (if any). This can be done by editing the specific tomcat xml files.
- obtain the fog.war file and/or rebuild it from source. It require ant to rebuild. It includes all necessary jar files for pdf and word support.
- Log in to the tomcat admin panel, and deploy the fog.war file.
- Log into the allure frontend, customize it for your local environment. Next add the bundle.zip file to the required packages (should see an error message about it).
- Through the tomcat admin panel, restart the fog deployment. One can use the command line and restar the entire tomcat install.
- create local accounts and enjoy using the system.

14.5 Test The System

14.5.1 Registration Testing

In the initial system, the registration window will show an image verification. This will show up correctly only if tomcat is allowing the war file correct local read/write permissions.

Attempt to registrer a new email and confirm an outgoing email has been sent and recieved. Monitor the catalina.out file for debug messages.

14.5.2 Verify account

Attempt to follow the verification email link and register a new account. Cataline.out will show any debug information.

14.5.3 Document Creation

The pdf document creation depends on key signing files and people.gzip file being present in the run directory. During the document creation process, one should see different taks being done by tailing the catalina.out file or checking the system logs.

14.5.4 Decoy pings

Create a decoy url and attempt to open it in a standard browser. You should recieve an alert email if the system is up and running correctly.

References

References

[1] Bell D. E. and LaPadula L. J., "Secure Computer Systems: Mathematical Foundations," MITRE Corporation, 1973.

[2] Bell, J. and Whaley, B. Cheating and Deception, Transaction Publishers, New Brunswick, NJ. 1982.

[3] Butler, J. and Sherri S., "Security: Spyware and Rootkits," Login, Vol 29, No 6, December 2004.

[4] Clark, D. D. and Wilson, D. R., "A Comparison of Commercial and Military Computer Security Policies," IEEE Symposium on Security and Privacy, pp. 184-194, 1987.

[5] Demers, A., Gehrke, J., Hong, M., Panda, B., Riedewald, M., Sharma, V., and White, W., "Cayuga: A General Purpose Event Monitoring System," CIDR, pp. 412-422, 2007.

[6] Detristan, T., Ulenspiegel, T., Malcom Y., and Von Underduk, M. S. "Polymorphic Shellcode Engine Using Spectrum Analysis," Phrack 11, 61-9, 2003.

[7] Friess, N., and Aycock, J.,"Black Market Botnets," Department of Computer Science, University of Calgary, TR 2007-873-25, July, 2007.

[8] Hoang, M. "Handling Today's Tough Security Threats," Symantec Security Response, 2006.

[9] The Honeynet Project. `http://www.honeynet.org`

[10] The Honeynet Project, "Know Your Enemy: Sebek, A Kernel based data capture tool," November, 2003.

[11] Honeypots. `http://www.honeypots.org/`

[12] Honeypot Mailing List, Security Focus.
`http://www.securityfocus.com/archive/119`

[13] Katz, John and Yehuda L., Introduction to Modern Cryptography, Chapman and Hall CRC Press, 2007.

[14] Kravets, D., "From Riches to Prison: Hackers Rig Stock Prices," Wired Blog Network, September, 2008.

[15] Krebs, B., "Web Fraud 2.0: Validating Your Stolen Goods," The Washington Post, August 20, 2008.

[16] Li, W., Stolfo, S. J., Stavrou, A., Androulaki, E., and Keromytis, A., "A Study of Malcode-Bearing Documents," DIMVA, pp. 231-250, 2007.

[17] Maloof, M. and Stephens, G. D., "ELICIT: A System for Detecting Insiders Who Violate Need-to-know," Recent Advances in Intrusion Detection (RAID), 2007.

[18] McRae, C. M. and Vaughn, R. B., "Phighting the Phisher: Using Web Bugs and Honeytokens to Investigate the Source of Phishing Attacks," Proceedings of the 40th Hawaii International Conference on System Sciences, 2007.

[19] Orbiscom. `http://www.orbiscom.com/`

[20] Richardson R., "CSI/FBI Computer Crime and Security Survey", 2007.

[21] Smith, R. M., "Microsoft Word Documents that Phone Home", Privacy Foundation, August, 2000.

[22] Song Y., Locasto M. E., Stavrou A., Keromytis A. D., and Stolfo S. J.. "On the infeasibility of modeling polymorphic shellcode," In Proceedings of the 14^{th} ACM conference on Computer and communications security (CCS07), pp. 541-551, 2007.

[23] Spitzner, L., "Honeypots: Catching the Insider Threat" Proceedings of ACSAC. Las Vegas, December, 2003.

[24] Spitzner, L., "Honeytokens: The Other Honeypot", Security Focus, 2003.

[25] Stoll, C. The Cuckoo's Egg, Doubleday, 1989.

[26] Symantec. Global Internet Security Threat Report, April 2008. Trends for July –December 07.

[27] Tomcat Apache. "http://tomcat.apache.org/", "visited on June, 2011".

[28] Webb, S., Caverlee, J., and Pu, C., "Social Honeypots: Making Friends with a Spammer Near You," In Proceedings of the Fifth Conference on Email and Anti-Spam (CEAS 2008), Mountain View, CA, August 2008.

[29] Ye, N., "Markov Chain Model of Temporal Behavior for Anomaly Detection," Proceedings of the 2000 IEEE Workshop on Information Assurance and Security, United States Military Academy, West Point, NY, pp. 171-174, June 2000.

[30] Yuill, J., D. Denning, and Feer, F., "Using Deception to Hide Things from Hackers : Processes, Principles, and Techniques," Journal of Information Warfare, 5(3):26-40, November, 2006.

[31] Yuill, J., Zappe M., Denning D., and Feer F.. "Honeyfiles: Deceptive Files for Intrusion Detection," Proceedings of the 2004 IEEE Workshop on Information Assurance, United States Military Academy, West Point, NY, pp. 116-122, June 2004.